

Patrones de diseño





Kevin R. Padilla Islas

jistro



kevin-padilla-islas



jistro



jistro




jistro.eth



**Cosas a
considerar**





1- Se tiene en consideración que el asistente ya tiene conocimientos en solidity básicos, sin embargo se tratará de explicar lo más simple

2- Cada término de patrón se hará una ronda de preguntas y (si es posible) una sección de hands on

3- No tengan miedo en preguntar, es un espacio seguro y abierto para todxs. No existen preguntas tontas.

4- Usaremos remix por simplicidad en los hand on

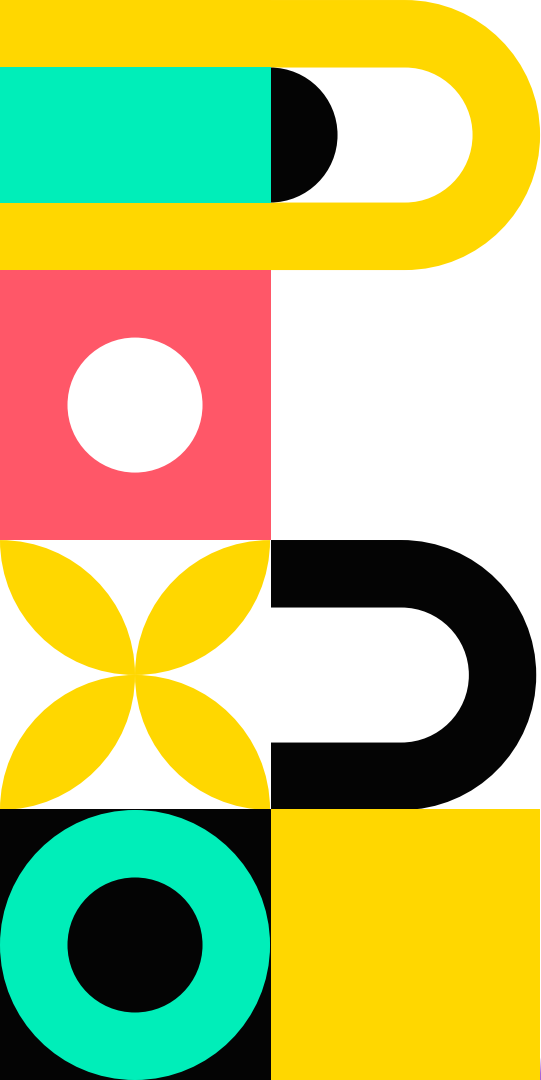


Patrones de diseño

Los patrones de diseño son soluciones para problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación.

Aunque nuestra aplicación sea única, tendrá partes comunes con otras aplicaciones. En lugar de reinventar la rueda, podemos solucionar problemas utilizando algún patrón, ya que son soluciones probadas y documentadas por multitud de programadores.

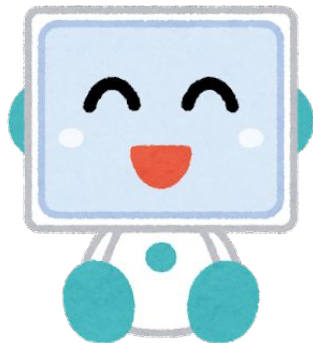




Porque usar patrones de diseño

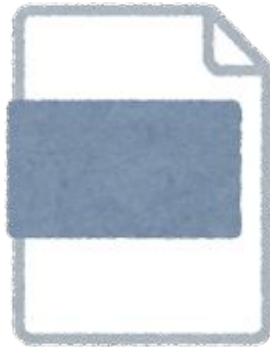
Mejora de la calidad de código

Los patrones de diseño son soluciones probadas y comprobadas para problemas comunes en la programación. Al utilizar estos patrones, puedes escribir código más limpio, estructurado y mantenible en Solidity.



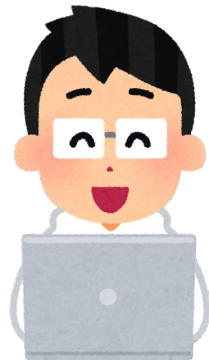
Reusabilidad

Los patrones de diseño fomentan la reutilización del código. Puedes utilizar los mismos patrones en diferentes partes de tu contrato inteligente o en proyectos diferentes, lo que ahorra tiempo y esfuerzo.



Facilita la comprensión

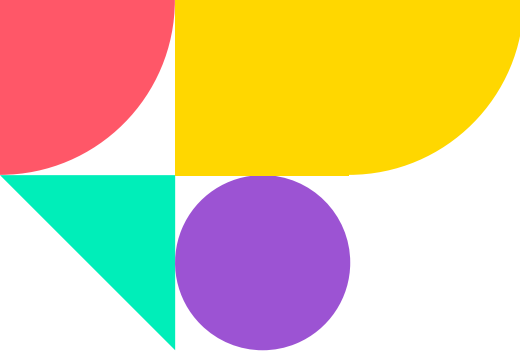
Los patrones de diseño son una especie de lenguaje común entre los desarrolladores. Cuando otros desarrolladores revisen tu código, será más fácil para ellos entender la estructura y el propósito de tu contrato inteligente si sigues patrones de diseño reconocidos.



Seguridad

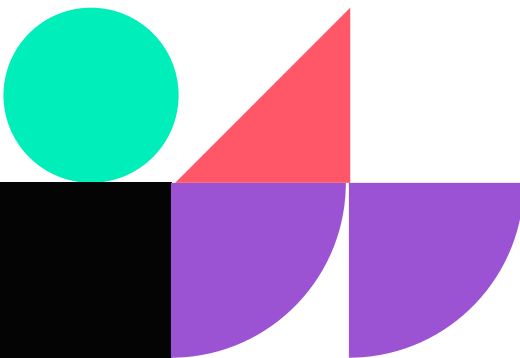
Algunos patrones de diseño están diseñados específicamente para mejorar la seguridad. Utilizar estos patrones puede ayudarte a evitar errores comunes que podrían exponer tu contrato a vulnerabilidades y ataques.





**Tipos de patrones
comunes**

—



01 Comportamiento

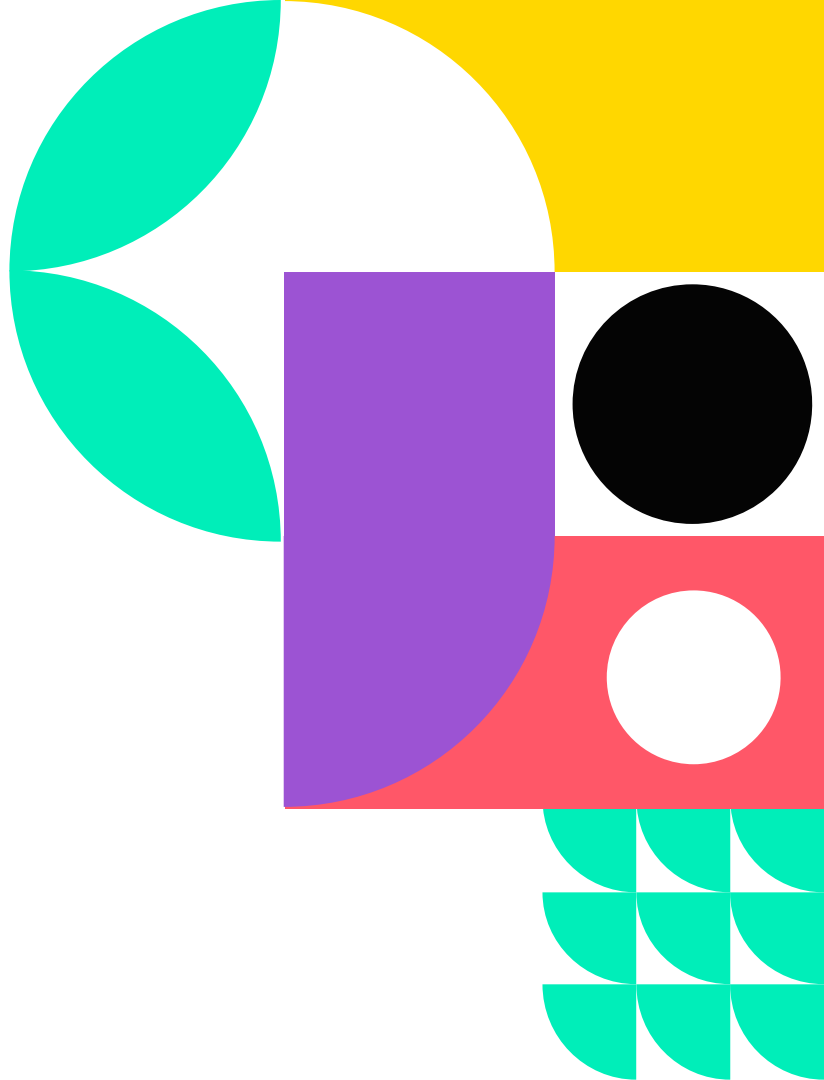
02 Seguridad

03 Mejora

04 Economicos

01

Patrón de comportamiento



Revisión de Guardias

- Los Contratos Inteligentes requieren una revisión de guardias.
- Su función principal es garantizar que los contratos operen de manera correcta.
- La ausencia de reguladores en blockchain hace que esta revisión sea crucial.
- En Solidity, se recomienda usar `require()` y `assert()` para implementar este patrón.



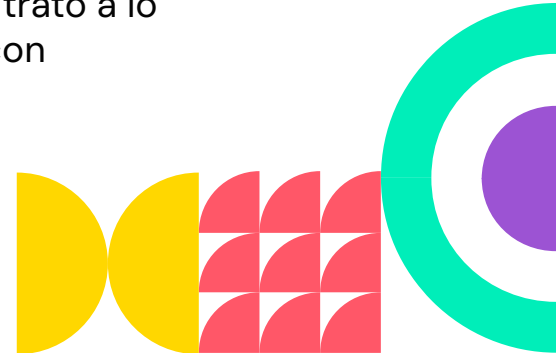
Cuándo y Cómo Aplicar este Patrón

- ¿Cuándo utilizar la Revisión de Guardias en contratos inteligentes?
 - Validar entradas de usuarios.
 - Verificar el estado del contrato antes de ejecutar la lógica.
- Comprobar invariantes en el código.
- Descartar condiciones no deseadas.
- Implementación recomendada en Solidity:
 - Evolución hacia el uso de `require()` para validar condiciones válidas.
 - Utilizar `assert()` para detectar errores internos.
 - Considerar las diferencias en el uso de gas entre `require()` y `assert()`.



Máquina de Estados

- El Patrón de Máquina de Estados se utiliza en contratos inteligentes para permitir que el contrato pase por diferentes etapas, cada una con funcionalidades específicas.
- Este enfoque es particularmente útil cuando un contrato debe adaptar su comportamiento a lo largo de su vida útil, como en el caso de subastas, juegos de azar o financiamiento colectivo.
- La motivación clave detrás de este patrón radica en la necesidad de gestionar el comportamiento del contrato a lo largo de su ciclo de vida de manera estructurada, con transiciones claras y controladas entre las etapas.



Cuándo aplicar este Patrón

- Cuando un contrato inteligente debe atravesar múltiples etapas durante su vida útil.
- Cuando se requiere que ciertas funciones del contrato sólo están accesibles en determinadas etapas.
- Cuando es esencial establecer transiciones de etapa claramente definidas y que no puedan ser prevenidas por los participantes.



Implementación y Consideraciones Importantes

- Para implementar este patrón:
 - Se pueden utilizar enumeraciones para representar las distintas etapas del contrato.
 - Se aplican modificadores de función para controlar el acceso a las funciones según la etapa actual.
- Consideraciones clave:
 - Las transiciones de etapa pueden ocurrir de diversas maneras, como mediante llamadas de función, transiciones temporizadas o cambios manuales por parte del propietario del contrato.
 - Es esencial llevar a cabo pruebas exhaustivas para garantizar que las transiciones de etapa sean seguras y predecibles en todas las circunstancias.



Patrón Oracle

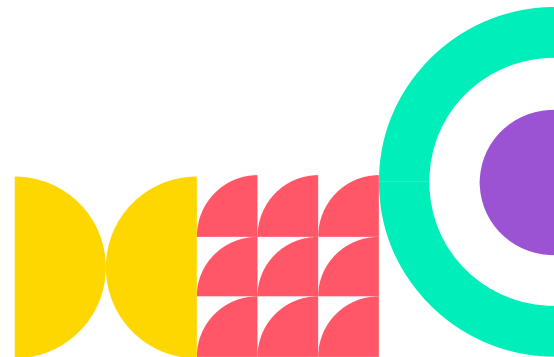
- El Patrón Oracle utiliza contratos inteligentes para obtener datos externos, convirtiéndolos en contratos híbridos.
- Necesario cuando un contrato requiere información externa, como eventos deportivos, tasas de cambio o datos en tiempo real.






Motivación y Necesidad de Oráculos

- La mayoría de los contratos en blockchain no pueden acceder directamente a datos externos debido a la descentralización y el consenso.
- Los oráculos funcionan como intermediarios que obtienen y proporcionan información externa a los contratos inteligentes.
- Se aplica cuando se necesita información externa confiable que no está disponible dentro de la cadena de bloques.



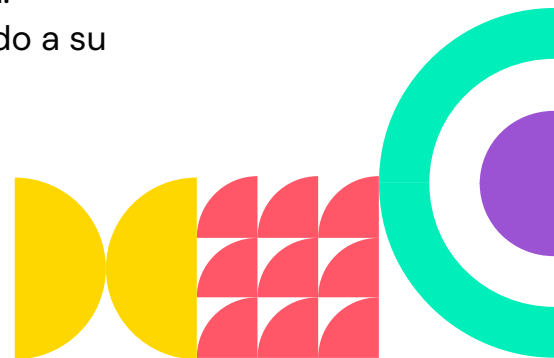


Implementación y Consecuencias del Patrón Oracle

- La implementación se enfoca en el contrato que solicita información, utilizando servicios de oráculos externos como Chainlink.
 - El contrato debe contar con métodos para enviar solicitudes al oráculo y recibir los resultados.
 - Consecuencias:
 - Habilita el acceso a datos externos, permitiendo casos de uso más complejos en contratos inteligentes.
 - Introduce un punto único de fallo y la necesidad de confiar en el oráculo y la fuente de datos externos.
- 

Patrón de Aleatoriedad

- El Patrón de Aleatoriedad busca generar números aleatorios en una cadena de bloques determinista, como Ethereum.
- Motivación: la necesidad de aleatoriedad en contratos inteligentes, especialmente en aplicaciones de juegos y apuestas.
- La cadena de bloques Ethereum es determinista y no puede generar números verdaderamente aleatorios de manera nativa.
- Desafíos: la aleatoriedad es difícil de lograr debido al consenso entre los mineros y a la naturaleza pública de la cadena.
- Se descartó el uso de sellos de tiempo de bloque debido a su manipulabilidad por parte de los mineros.



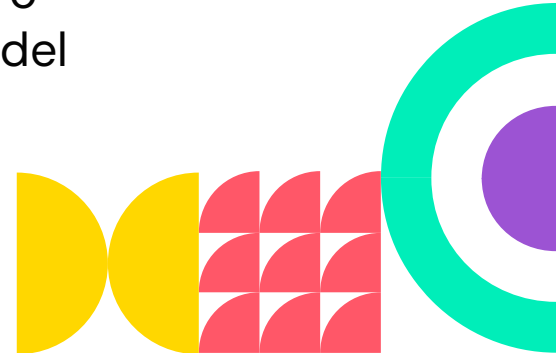
Enfoques para la Generación de Aleatoriedad y cuando aplicar

- Diferentes enfoques incluyen:
 - Block hash PRNG: Utilización del hash de un bloque como fuente de aleatoriedad.
 - Oracle RNG: Obtención de aleatoriedad de un oráculo externo, como se describe en el patrón Oracle.
 - Collaborative PRNG: Generación colaborativa de números aleatorios dentro de la cadena de bloques.
- Se aplica cuando:
 - Se requiere aleatoriedad impredecible.
 - No se desean servicios externos para la aleatoriedad.
 - Se confía en una entidad de confianza para proporcionar semillas de aleatoriedad.




Implementación y Consecuencias

- Implementación: una entidad de confianza proporciona una semilla que se combina con el hash de un bloque futuro para generar un número pseudo random.
- Consecuencias: la aleatoriedad es pseudorandom, segura y con costos controlados. Sin embargo, existe un ligero retraso entre la solicitud y la obtención del número aleatorio.





Comparación con el Patrón Oracle

- En comparación con el patrón Oracle que puede proporcionar aleatoriedad verdadera mediante servicios externos, el enfoque de aleatoriedad es más adecuado para contratos simples con bajo valor financiero.
 - Contratos más complejos o que manejan apuestas más grandes tienden a utilizar servicios de oráculos para acceder a números aleatorios.
- 

The top-left corner features a purple quarter-circle and a yellow and red striped vertical bar. The top-right corner features a teal concentric circle with a purple center.

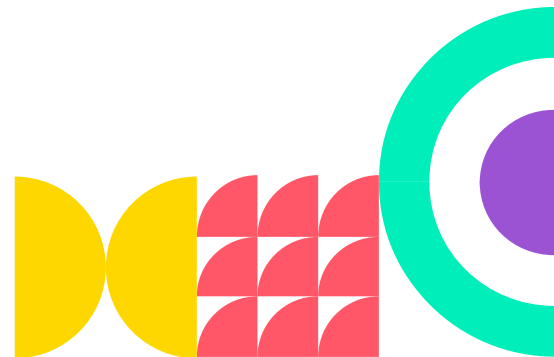
02

Patrones de seguridad

The bottom-left corner features a yellow circle, a black circle, and a teal circle. The bottom-right corner features a teal grid of semi-circles.

Patrón de Restricción de Acceso

- El Patrón de Restricción de Acceso limita el acceso a la funcionalidad del contrato según criterios específicos.
- Utilizado en entornos de cadena de bloques para controlar quién puede interactuar con ciertas funciones del contrato.



Aplicabilidad del Patrón

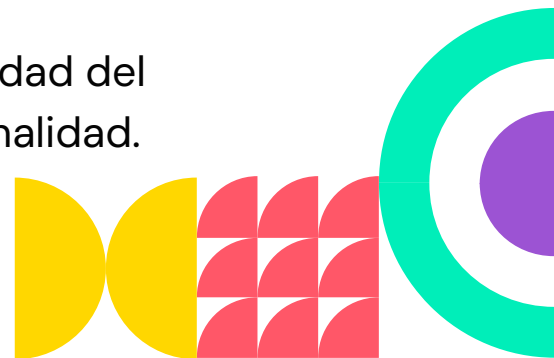
- Utiliza el patrón de Restricción de Acceso cuando:
 - Requieren que ciertas funciones sólo sean llamadas en circunstancias específicas.
 - Deseas aplicar restricciones similares a varias funciones en el contrato.
 - Buscas aumentar la seguridad del contrato inteligente contra accesos no autorizados.





Consecuencias del Patrón

- Consideraciones importantes:
 - Impacto en la legibilidad del código al indicar restricciones en la cabecera de la función.
 - Posible complejidad en el seguimiento del flujo de ejecución.
- Ventajas:
 - Adaptabilidad a diversas situaciones.
 - Alta reutilización, simplificando la seguridad del contrato al limitar el acceso a su funcionalidad.



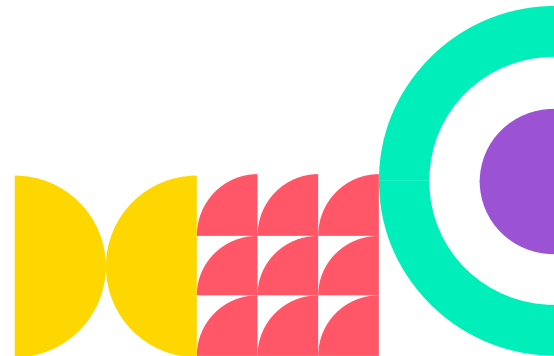
Transferencia Segura de Ether

- El patrón "Transferencia Segura de Ether" se centra en transferir ether de manera segura desde un contrato a otra dirección en Ethereum.
- Motivación: garantizar transferencias seguras de ether y adaptarse a la evolución de los métodos en Solidity
- En el pasado, se utilizaban métodos como `<dirección>.send(cantidad)` con limitaciones en gas y manejo de excepciones.
- Introducción de `transfer(cantidad)` como una mejora, pero sin especificación de cantidad de gas.
- Existe `call.value`, más versátil pero menos seguro debido a su flexibilidad.




Resumen y Desafíos

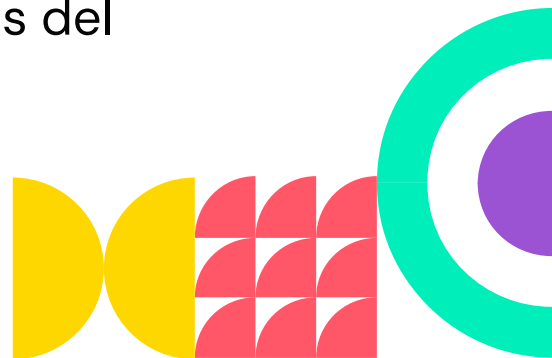
- En resumen, el patrón "Transferencia Segura de Ether" ofrece flexibilidad pero también plantea desafíos en la elección del método correcto para transferir ether de manera segura.
- La seguridad y la reversión automática en caso de error son factores clave a considerar en la elección del método.





Patrón Pull over Push

- El patrón "Pull over Push" se enfoca en transferir el riesgo asociado con la transferencia de ether en Ethereum desde el contrato hacia el usuario.
 - Motivación: Evitar posibles fallas en las llamadas externas al enviar ether y prevenir que un fallo afecte a otras operaciones del contrato.
- 



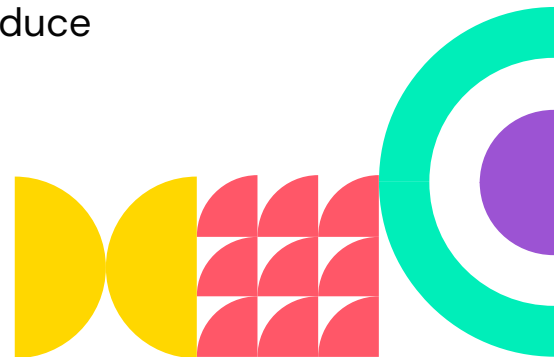
Mecanismo, Aplicaciones y Utilidad del Patrón

- En lugar de enviar ether automáticamente a los usuarios, el patrón permite que los usuarios soliciten activamente la retirada de fondos.
- Utiliza un mapeo para llevar un registro de los saldos pendientes de los usuarios, aislando cada transferencia y reduciendo el riesgo de afectar a otras operaciones del contrato.
- El patrón es útil cuando se deben manejar múltiples transferencias de ether con una sola llamada de función.
- Adecuado para evitar riesgos asociados con transferencias, pero puede aumentar la complejidad para los usuarios, ya que deben realizar una transacción adicional para solicitar la retirada de fondos.



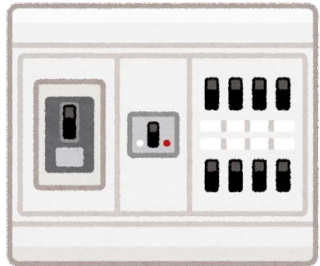
Patrón 'Circuit Breaker'

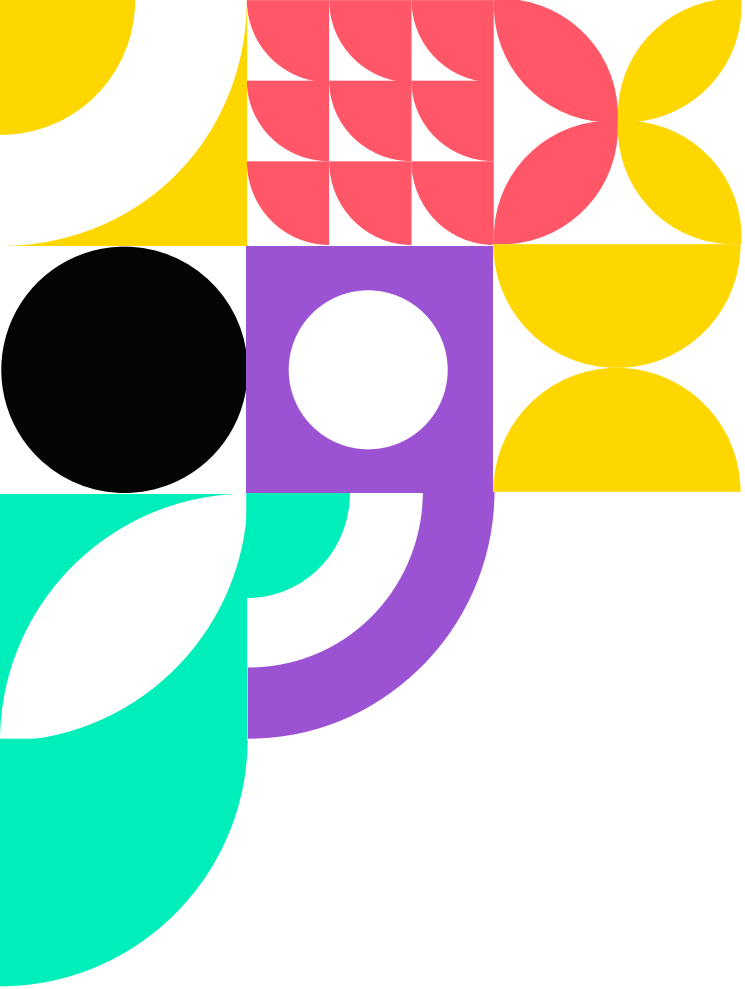
- El patrón "Circuit Breaker" es un enfoque de diseño que permite pausar o desactivar la funcionalidad de un contrato inteligente en respuesta a problemas o vulnerabilidades imprevistos.
- Permite que un administrador designado active o desactive este interruptor.
- Protege el contrato y sus usuarios de posibles daños.
- Ofrece una respuesta inmediata a vulnerabilidades, protege contra ataques de préstamos flash y reduce pérdidas económicas.



Desafíos, Consideraciones y Beneficios

- Desafíos incluyen preocupaciones sobre la centralización y la confianza en los administradores.
- Implementación: diseño del interruptor, definición de condiciones de emergencia, establecimiento de control de acceso, pruebas, auditorías y documentación.
- El patrón "Circuit Breaker" es crucial para garantizar la seguridad y estabilidad de los contratos inteligentes en blockchain.
- Brinda una respuesta efectiva a problemas inesperados y protege a los usuarios.





Patrón de mejora

03



Patrón Proxy Delegate

- El patrón Proxy Delegate es una estrategia para permitir actualizaciones de contratos sin afectar dependencias existentes en el contexto de contratos inteligentes en Ethereum.
- Ethereum tiene contratos inmutables, lo que significa que una vez desplegados, no pueden cambiar.
- Se necesita un mecanismo para actualizar contratos sin romper dependencias existentes.



Componentes Clave del Patrón

- Proxy (Proxy): Actúa como intermediario entre usuarios y el contrato lógico real.
- Delegado (Delegate): Contiene la lógica y funciones del contrato original. Puede actualizarse sin afectar al Proxy.
- El Proxy delega llamadas de función al Delegado.
- El mensaje `delegatecall` permite al Proxy pasar llamadas de función al Delegado sin conocer la firma de la función.
- El código en la dirección de destino se ejecuta en el contexto del contrato llamador (Proxy).





Registro y Limitaciones

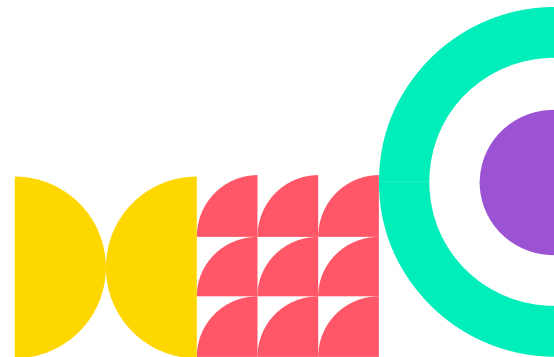
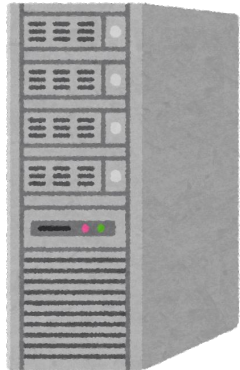
- El Proxy puede mantener un registro de la dirección actual del Delegado para rastrear la versión.
- Limitaciones: El Delegado debe seguir una estructura de almacenamiento específica para permitir la adición de campos, pero no la eliminación ni el cambio de campos existentes.
- Los usuarios deben confiar en que las actualizaciones no introducirán comportamientos no deseados o maliciosos.
- Estrategias de actualización controlada y períodos de prueba pueden mitigar la preocupación de los usuarios.





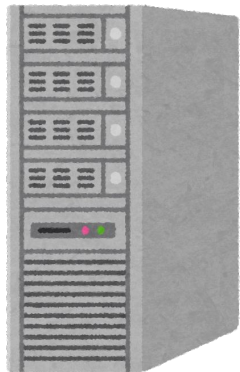
Patrón de Almacenamiento Eterno

- El patrón de Almacenamiento Eterno aborda la necesidad de mantener datos después de una actualización del contrato en contratos inteligentes de Ethereum.
- Conservar datos importantes, como información de usuarios o saldos de cuentas, al desplegar nuevas versiones de contratos en la red es un desafío.
- Migrar datos sería costoso y complejo.



Participantes Clave

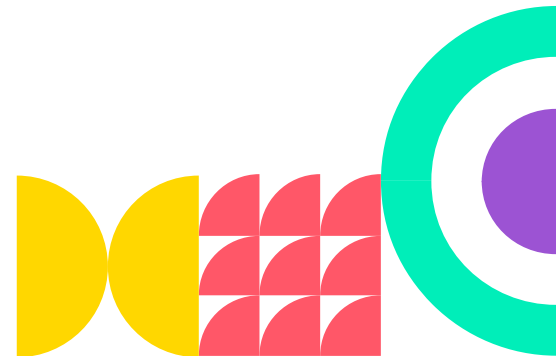
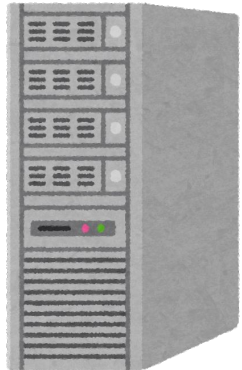
- Contrato Almacenamiento Eterno: Almacena datos de manera persistente y actúa como almacén de datos independiente.
- Propietario/Administrador: Gestiona y actualiza la dirección de la última versión del contrato que utiliza el Almacenamiento Eterno.
- Contrato que Requiere Almacenamiento: Necesita acceder y utilizar datos almacenados en el Almacenamiento Eterno.





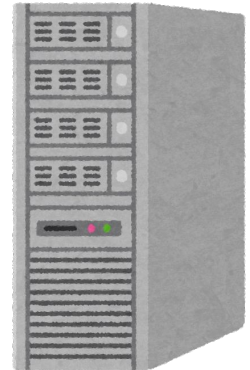
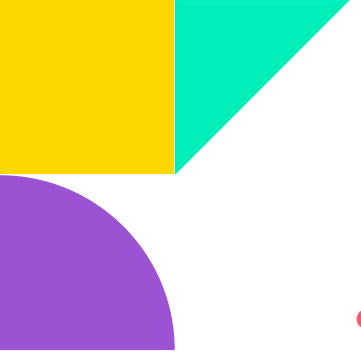
Funciones y Autorización

- Funciones para establecer, obtener y eliminar valores en el Almacenamiento Eterno.
- Estas funciones solo pueden ser llamadas desde la última versión del contrato, garantizado mediante el uso de un modificador.



Consecuencias del Patrón

- Ventajas: Elimina la necesidad de migración de datos después de actualizar un contrato, simplifica las actualizaciones y garantiza la persistencia de datos importantes.
- Desventajas: Introduce complejidad debido a llamadas externas y uso de funciones hash. Puede influir en la confianza de los usuarios debido a la capacidad de cambiar los contratos, lo que va en contra de la inmutabilidad de la cadena de bloques.





Patrones economicos


02





Importante

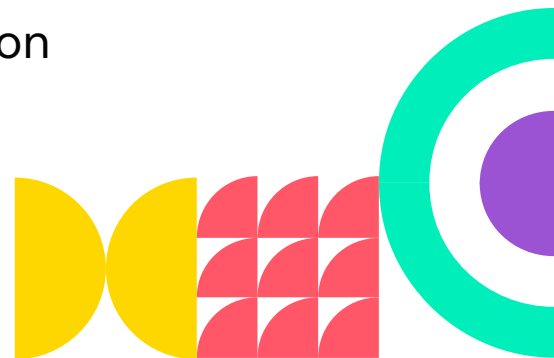
Patrones económicos significa ahorrar en gas mas no tiene que ver
con protocolos DeFi





Patrón de Comparación de Igualdad de Cadenas

- Este patrón se centra en verificar si dos cadenas son iguales de manera eficiente en términos de gas en Solidity.
- Comparar cadenas en otros lenguajes de programación es simple, pero en Solidity, no existe una funcionalidad eficiente para hacerlo.
- Varias soluciones existen, pero algunas consumen mucho gas, especialmente con cadenas largas.



Aplicabilidad

- Utiliza este patrón cuando desees verificar si dos cadenas son iguales.
- Es especialmente eficiente para cadenas con más de dos caracteres.
- Minimiza el consumo promedio de gas para una variedad de cadenas.
- Principalmente se utiliza en contratos inteligentes y bibliotecas internas.
- Participantes principales: la función que implementa la comparación y la función que realiza la solicitud.



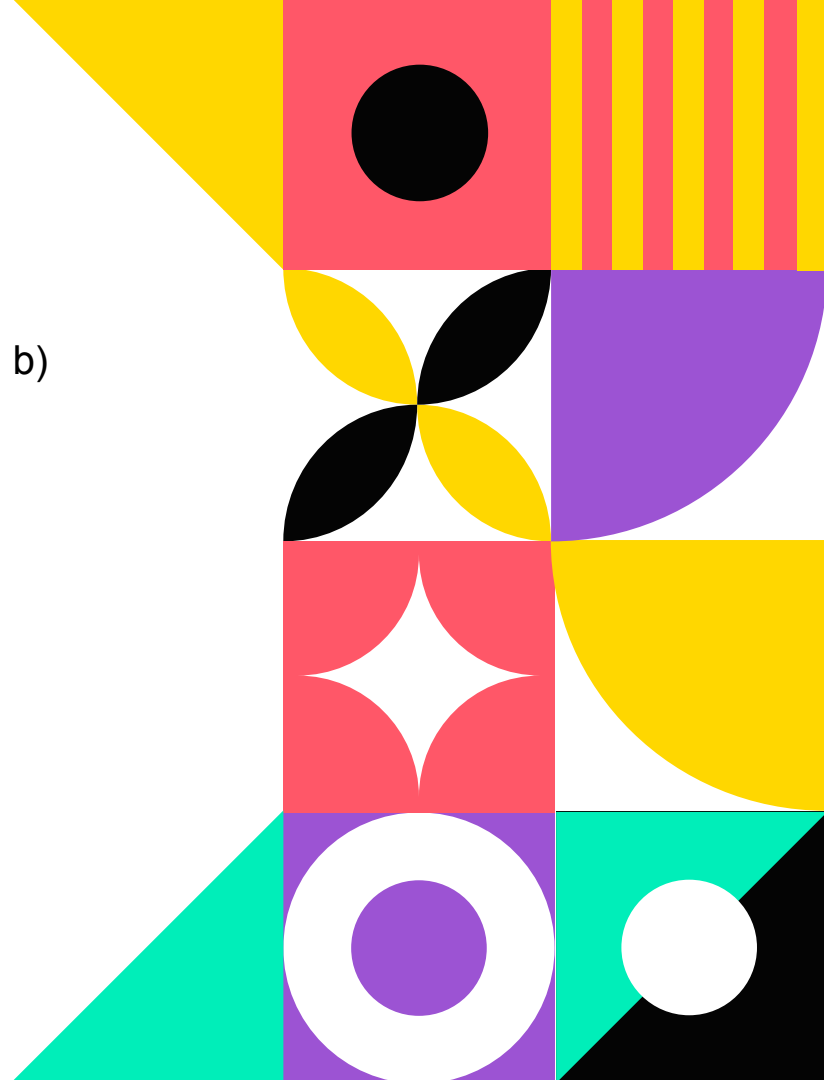


Implementación

- Verifica si las dos cadenas tienen la misma longitud.
- Convierte las cadenas al tipo de datos "bytes" para obtener la longitud.
- Descarta pares de cadenas con longitudes diferentes para ahorrar gas.
- Aplica la función hash criptográfica keccak256() a ambas cadenas.
- Compara los hashes para verificar la igualdad completa de las cadenas.



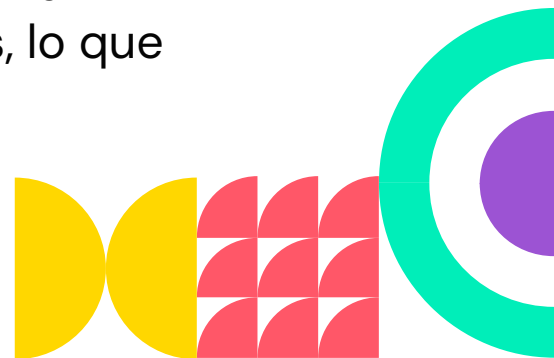
```
function hashCompareWithLengthCheck(string a, string b)
internal returns (bool) {
    if(bytes(a).length != bytes(b).length) {
        return false;
    } else {
        return keccak256(a) == keccak256(b);
    }
}
```





Patrón de Empaquetamiento Ajustado de Variables

- Este patrón tiene como objetivo optimizar el consumo de gas al almacenar o cargar variables de tamaño fijo en Solidity.
- La EVM (Ethereum Virtual Machine) asigna almacenamiento en Ethereum de manera particular.
- Variables de tamaño fijo se almacenan una detrás de la otra en ranuras de 32 bytes, lo que reduce el consumo de gas.



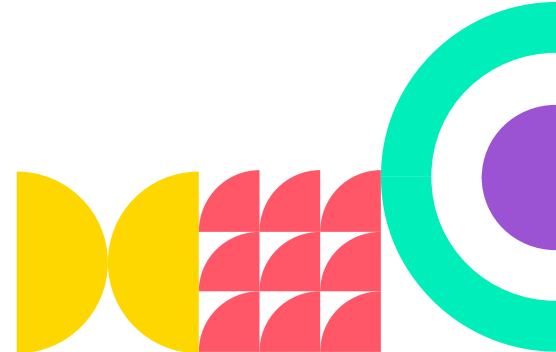
Aplicabilidad e Implementación

- Se aplica a variables de estado, dentro de estructuras (structs) y a matrices de tamaño fijo.
- Utiliza el tipo de dato más pequeño posible para garantizar la ejecución correcta del código.
- Agrupa todas las variables que deben ir juntas en una ranura de almacenamiento.
- Declara las variables una después de otra en el código.
- La EVM combina múltiples lecturas o escrituras en una sola operación, ahorrando gas.



Consideraciones

- Aumento en la complejidad: El reordenamiento de variables puede aumentar la complejidad del código.
- Pérdida de legibilidad: El código puede volverse menos legible debido a la reorganización de variables.





Patrón de Construcción de Matrices en Memoria

- Este patrón se enfoca en la eficiente agregación y recuperación de datos en el almacenamiento del contrato en Ethereum.
- Este patrón es aplicable cuando deseas agregar y leer datos del almacenamiento del contrato de manera eficiente.
- Es especialmente útil para recuperar datos de múltiples fuentes sin incurrir en costos de gas innecesarios.



Implementación

- Utiliza un tipo de datos fácil de iterar, como una matriz (array), para almacenar los datos en el contrato.
- Utiliza una estructura de datos personalizada (struct) para representar los atributos de los elementos de datos.
- Los datos se almacenan en el contrato, pero cuando se consulta, se crea una matriz en la memoria en lugar de almacenarla en el almacenamiento del contrato.
- La operación se realiza en una función "view", lo que significa que no modifica el estado del contrato y no incurre en costos de gas.
- Al llamar a la función desde otro contrato, se requerirá una transacción y se consumirá gas.



Gracias!

Si tienes alguna pregunta no dudes en hablar c:



CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**